VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

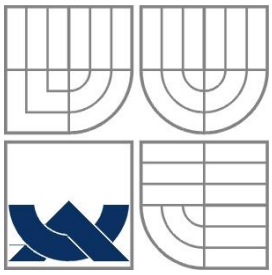
IMPLEMENTACE ALGORITMŮ ZALOŽENÝCH NA ROZHODOVACÍCH STROMECH V JAZYCE C#

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

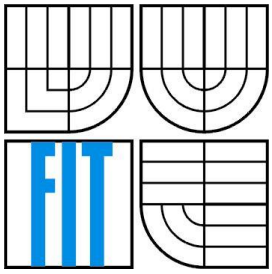
AUTOR PRÁCE
AUTHOR

LUKÁŠ GROLIG

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

IMPLEMENTACE ALGORITMŮ ZALOŽENÝCH NA ROZHODOVACÍCH STROMECH V JAZYCE C#

IMPLEMENTATION OF ALGORITHMS BASED ON DECISION TREES IN C#

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ GROLIG
AUTHOR

VEDOUCÍ PRÁCE — ING. ROSTISLAV STRÍŽ
SUPERVISOR

BRNO 2013

Abstrakt

Tato bakalářská práce se zabývá výběrem dolovacích algoritmů pro klasifikaci založených na rozhodovacích stromech pro analytický systém vyvíjený v rámci projektu Systém pro zvýšení bezpečnosti v prostředí Internetu analýzou šíření škodlivého kódu. U vybraných algoritmů je popsána jejich implementace v jazyce C#. Implementované algoritmy jsou následně testovány z hlediska rychlosti učení algoritmu a přesnosti klasifikace. Na základě výsledků experimentů jsou sepsány závěry a dána doporučení pro uživatele těchto algoritmů.

Abstract

This bachelor thesis is focused on selection of data mining algorithms based on decision trees for an analytical system developed under the project System for the Internet security increase based on malware spreading analysis. Selected algorithms are described in greater detais, as well as their implementation in the C# language. These algorithms are then tested with regards to their training speed and classification accuracy. Finally, this thesis presents further conclusions and recommendations based on performed experiments.

Klíčová slova

Dolování z dat, dolovací algoritmy, klasifikace, rozhodovací stromy, random forest, sprint

Keywords

Data mining, mining algorithms, classification, decision trees, random forest, sprint

Citace

Grolig Lukáš: Implementace algoritmů založených na rozhodovacích stromech v jazyce C#, bakalářská práce, Brno, FIT VUT v Brně, 2013

Implementace algoritmů založených na rozhodovacích stromech v jazyce C#

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Rostislava Stríže
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Grolig
14. 5. 2013

Poděkování

Děkuji vedoucímu práce Ing. Rostislavu Strížovi za konzultace, které mi pomohly při výběru algoritmů a vylepšení jejich implementace, která díky tomu zvýšila přesnost výsledků.

Obsah

1 Úvod ... 2
2 Dolování z dat ... 3
2.1 Proces získávání znalostí z databází ... 3
3 Dolovací algoritmy ... 5
3.1 Rozdělení algoritmů dle účelu ... 5
3.1.1 Klasifikační algoritmy ... 5
3.1.2 Regresní algoritmy ... 6
4 Výběr algoritmů pro implementaci ... 7
4.1 Srovnání algoritmů ... 7
4.2 Zhodnocení výsledků srovnání ... 8
5 Algoritmy založené na rozhodovacích stromech ... 9
5.1 SPRINT ... 12
5.2 RANDOM FOREST ... 13
6 Analytický systém ... 15
7 Implementace vybraných algoritmů ... 16
7.1 API analytického systému ... 16
7.2 Datové struktury používané pro algoritmy ... 17
7.2.1 Datová struktura uzlu ... 17
7.2.2 Datová struktura stromu ... 18
7.2.3 Datová struktura lesa ... 19
7.3 SPRINT ... 19
7.4 RANDOM FOREST ... 24
8 Experimenty s implementovanými algoritmy ... 26
8.1 Body v rovině ... 26
8.2 Body v prostoru ... 26
8.3 Přeživší na Titanicu ... 27
8.4 Hodnocení přesnosti algoritmů ... 28
8.5 Zhodnocení rychlosti algoritmů ... 29
8.6 Vliv nastavení na výsledky ... 29
8.6.1 Hloubka stromu ... 29
8.6.2 Velikost lesa ... 29
8.6.3 Počet kandidátů pro dělení ... 29
9 Závěr ... 30
Literatura ... 31

1 Úvod

V dnešním světě informačních technologií generují počítače velké množství dat, které je pro člověka nemožné zpracovat a vyhodnotit. Z toho důvodu začaly vznikat systémy sloužící pro uložení těchto dat a zároveň také systémy schopné data analyzovat, zpracovat a podat nám informace, které následně můžeme využít v náš prospěch. Oblast zpracování dat prochází v posledních letech velmi intenzivním vývojem, který nám přináší nové efektivnější algoritmy nebo modifikace stávajících, podávající výsledky rychleji a přesněji.

V kapitole 3 práce shrnuje obecné informace o data miningu a získávání znalostí z databází. Cílem této práce bylo vybrat klasifikační algoritmy založené na rozhodovacích stromech a ty implementovat. V kapitole 4 jsou zkoumána dostupná řešení a proveden výběr algoritmů pro implementaci. Vybrané algoritmy jsou popsány v kapitole 5 a na to je navázána ukázka zajímavých částí implementace v kapitole 6. V kapitole 7 je u vybraných algoritmů popsána jejich implementace a demonstrována funkčnost na testovací množině dat.

Kapitola 8 demonstruje možnosti nastavení parametrů algoritmů a to, jakým způsobem ovlivní výsledky.

Výsledkem je zásuvný modul pro projekt „Systém pro zvýšení bezpečnosti v internetu analýzou šíření škodlivého kódu“, který je stručně popsán v kapitole 6.

2 Dolování z dat

Dolováním z dat (anglicky *data mining*) se nazývá proces extrakce zajímavých (netriviálních, skrytých, dříve neznámých a potenciálně užitečných) modelů dat a vzorů z velkých objemů dat. Tyto modely a vzory reprezentují znalosti získané z dat. [1]

Tyto vzory se nedají získat prostým SQL dotazem nad databází (netriviálnost). Stejně tak nejsou vidět na první pohled (skrytost) a musíme tedy použít nějaký sofistikovaný přístup k jejich získání.

Příkladem může být velký internetový obchod nabízející zboží různých kategorií. Abychom navýšili obrat a zisk, je třeba nabídnout zákazníkům produkty, které potřebují nebo po nich touží. Z tohoto důvodu začneme sledovat položky, které si konkrétní zákazník prohlíží, porovnává, vkládá do košíku apod. Někteří ze zákazníků si zboží poté objednají. V tuto chvíli můžeme vidět, že zákazníci si většinou k produktu koupí ještě jiný související. Právě na základě těchto informací je možné zákazníky začít rozdělovat do skupin a konkrétní cílové skupině nabídnout kombinace produktů, které potřebují, nebo je do této chvíle nenapadlo, že by potřebovali. Dále můžeme začít posílat místo obecných emailových nabídek konkrétní nabídku na produkty odpovídající těm, které náš zákazník prohlížel. Toto vše nám ve výsledku přinese užitek, což je hlavním důvodem, proč se začít dolováním z dat zabývat.

Jiným příkladem může být použití v medicíně. Ve snímcích z CT (počítačová tomografie) hlavy lékaři označí část, kde se nachází nádor na mozku. Vznikne databáze vzorků, které je možno použít pro počítačové učení. Výsledkem tohoto učení je model, díky kterému je program schopný v neznámých neoznačených snímcích sám nalézt onemocnění.Podobných příkladu můžeme ve světě najít spousty – od podpory podnikových procesů až po medicínu.

Jak bylo zmíněno, výsledkem dolování jsou modely anebo vzory. Je na uživatelích (analyticích) k jakému účelu je použijí. Je možné je zařadit do programu, který se bude starat o predikci (předpověď) nebo je vhodným způsobem vizualizovat, vyhodnotit a dále využít získané poznatky.

2.1 Proces získávání znalostí z databází

Proces získávání znalostí z databází (anglicky knowledge discovery in databases) je proces získávání užitečných informací z kolekcí dat. Je to jedna z široce používaných technik pro dolování z dat. [2]

Nyní se podíváme, jak proces získávání znalostí z databáze vypadá, a z jakých se sestává kroků:

1. *Čištění dat* – cílem je vypořádat se s chybějícími daty, odstranit šum a nekonzistence, jelikož by mohli způsobit znehodnocení modelu.
2. *Integrace dat* – cílem je spojení dat pocházejících z několika zdrojů. Často se integrace a čištění dat provádí společně. Jednak proto, že vyčištěná data potřebujeme někam ukládat, jednak proto,

že jedním ze zdrojů nekonzistence jsou typicky data pocházející z více zdrojů. V takovém případě jsou data ukládána do datového skladu.

3. *Výběr dat* - cílem je vybrat data, která jsou pro řešení dané analytické úlohy relevantní. Pokud získáváme znalosti z dat uložených v relační databázi, pracujeme typicky s konzistentním homogenním zdrojem (např. tabulkou). V tomto kroku tedy vybereme z tabulky relevantní sloupce. V případě datového skladu můžeme analogicky vybírat dimenze.
4. *Transformace dat* - cílem je transformovat data do konsolidované podoby vhodné pro dolování. Může jít například o sumarizaci nebo agregaci. V případě použití datového skladu pro dolování může transformace předcházet výběru dat, protože může být součástí tvorby datového skladu.
5. *Dolování dat* - jádro procesu získávání znalostí, jehož cílem je aplikací určité metody a konkrétního algoritmu extrahovat z dat vzory, resp. vytvořit model dat.
6. *Hodnocení modelu a vzorů* - cílem je identifikovat skutečně zajímavé vzory pomocí měr užitečnosti.
7. *Prezentace znalostí* - cílem je prezentovat výsledky dolování uživateli využitím technik vizualizace a reprezentace znalostí.

V této kapitole jsme prošli nejdůležitější informace týkající se dolování z dat. Ty by měly čtenáři poskytnout představu o čem dolování dat je a ve kterých oblastech jej můžeme využít. Dále se tato práce zabývá hlavně krokem 6 a 7.

Více informací o procesu získávání znalostí z databází naleznete v [1].

3 Dolovací algoritmy

V této kapitole se podíváme na základní proces dolování dat a rozdělíme si algoritmy do kategorií dle úkolů, pro které bychom je chtěli využít. Úkolem dolovacích algoritmů je v datové množině najít určité vzory a vytvořit model dat. Způsob jakým toho algoritmy docílí, závisí na jejich typu a implementaci.

Samotný proces dolování můžeme rozdělit do několika částí:

1. *Trénink na trénovací množině* – než můžeme začít data využívat, je třeba nejprve vytvořit jejich model. Fázi, kdy se tvoří znalostní model, nazýváme trénovací fáze. Nejprve si připravíme tréninkovou množinu dat. Z té odstraníme šum a řádky s chybějícími hodnotami (případně dopočteme jejich hodnotu např. získáním průměrné hodnoty aj.). Když máme připravená data, spustíme nad nimi dolovací algoritmus, který postupným učením (bude probráno v následujících kapitolách) vytvoří prediktivní model.
2. *Testování modelu* – pro vytvořený model musíme provést jeho vyhodnocení. To znamená, že model otestujeme nad známými daty a budeme sledovat, s jakou přesností predikuje výsledky. Pro získání testovací množiny často používáme přístup, při kterém rozdělíme předzpracovanou počáteční množinu dat na množinu trénovací a testovací. Postupně zaznamenáváme výsledky – kolikrát výsledek odpovídal očekávanému – a z nich sestavíme statistiku, ze které zjistíme procentuální úspěšnost modelu. V případě, že nedosáhneme požadovaných výsledků, musíme provést proces dolování od začátku např. s jinými parametry nebo jinou trénovací množinou. Znovu tedy připravíme trénovací a testovací data a spustíme algoritmus. Pokud se požadované výsledky stále nedostavují, je třeba zvolit jiný algoritmus.
3. *Nasazení modelu* – pokud model splnil počáteční požadavky, můžeme jej dále využít. Běžně se nabízí dva způsoby využití. První je vizualizace modelu, která nám poskytne informace o vazbách a informace, na kterých můžeme založit naše rozhodnutí, nebo rozšířit naše znalosti týkající se problému. Druhým způsobem je nasazení v rámci programu, který se bude starat o automatické vyhodnocení neznámých dat. Je vždy na uživateli k jakému účelu výsledek použije.

3.1 Rozdělení algoritmů dle účelu

Algoritmy můžeme rozdělit dle jejich využití do několika skupin, z nichž nejpoužívanější jsou probrány dále.

3.1.1 Klasifikační algoritmy

Cílem klasifikačních algoritmů je automaticky identifikovat a přiřadit datům ze vstupní množiny třídu, do které náleží. Výsledek, který chceme získat, nabývá diskrétních hodnot.

Oblíbené techniky pro klasifikaci jsou např. SVM (suport vector machine), Bayesovská klasifikace nebo rozhodovací stromy – více ve [2] a [3].

Jako příklad můžeme uvést automatické vyhodnocení paketů, které prochází přes firewall. Každý pak má určité atributy na základě, kterých můžeme určit, zda se jedná o útok nebo ne. Pakety, které jsou klasifikovány jako nebezpečné, můžeme zahodit.

3.1.2 Regresní algoritmy

Regresní algoritmy (někdy také prediktivní) se snaží předpovědět hodnotu určitého atributu. Tento atribut nabývá spojitých hodnot. Příkladem může být ohodnocení očekávané ceny akcie k určitému datu na základě jejích atributů a podobných scénářů.

Dále se budeme zabývat pouze algoritmy z kategorie klasifikačních algoritmů založených na rozhodovacích stromech.

4 Výběr algoritmů pro implementaci

Před samotnou implementací algoritmů byla provedena analýza dostupných řešení. Tato analýza sloužila pro výběr algoritmů, které byly v rámci práce implementovány. Bylo třeba stanovit kritéria, na základě kterých, se provedl výběr. Kritéria pro porovnání byla převzata ze zdroje [1].

Nejdůležitějším kritériem byla *přesnost předpovědi*. Ta říká, v kolika procentech daný model správně klasifikuje „nová“ data, tedy data, která nebyla obsažena v trénovací množině. Druhým směrodatným kritériem je *rychlost*, které představuje výpočetní složitost pro vygenerování a používání klasifikačních pravidel. Je vyjádřena časem potřebným k vytvoření znalostního modelu a predikci trénovací množiny. Podmínkou zadavatele práce je schopnost algoritmu se vypořádat s velkým množstvím dat. Tuto schopnost popisuje *stabilita*.

Ostatní možná kritéria, která při výběru hrála vedlejší roli, jsou:

Robustnost, tj. schopnost vytvořit správný model, pokud daná množina obsahuje šum a chybějící hodnoty. Toto kritérium je pro nás vedlejší, jelikož se o šum a chybějící hodnoty postaráme před samotným spouštěním algoritmu a nebude nás tedy zajímat, jak si s tím algoritmus dokáže poradit. Posledním kritériem dle zdroje je *interpretovatelnost*, která udává, jak jednoduchá (resp. obtížná) je orientace ve znalostním modelu (např. vizualizovaném).

4.1 Srovnání algoritmů

Pro srovnání byl nejprve udělán průzkum používaných algoritmů pro klasifikaci založených na rozhodovacích stromech. Tento průzkum znamenal projít odborné zdroje a vyhledat dostupná řešení. Výsledný seznam pak vypadal následovně:

ID3, C4.5, CART, CHAID, MARS, BEST FIRST TREE, SLIQ, SPRINT, RANDOM FOREST

Při jejich srovnání bylo využito existujících implementací v některém z frameworků určeném pro dolování dat. Jako ideální pro výchozí srovnání se jevil framework *Weka*. Experimenty a jejich vyhodnocení poskytl zdroj [4].

V rámci srovnání jsou porovnávány algoritmy ID3, C4.5, CART, BFT, SLIQ, SPRINT a RF. Všechny tyto algoritmy byly testovány na několika sadách dat obsahujících větší počet atributů (17 - 37), různé množství klasifikačních tříd (2 - 26) a počtem vzorků mezi 1 000 – 10 000.

4.2 Zhodnocení výsledků srovnání

Ze srovnání [4] vyšel nejlépe algoritmus Random Forest, který je předveden v kapitole 5.2. Dosáhl nejlepších výsledku v rámci přesnosti klasifikace. Vlivem specifik ale nedosáhl nejlepších výsledků v rychlosti. To je dáno částečně principem algoritmu, ale hlavně jeho implementací[1].

Dle srovnání se jako vhodní kandidáti na druhý algoritmus jevily algoritmy C4.5 a algoritmus SPRINT. SPRINT dosahoval mírně lepších výsledků v přesnosti, C4.5 dokázal rychleji sestavit model a následně provést klasifikaci. O druhém algoritmu tedy rozhodl způsob implementace. Byla zvolena méně běžná varianta – algoritmus SPRINT.

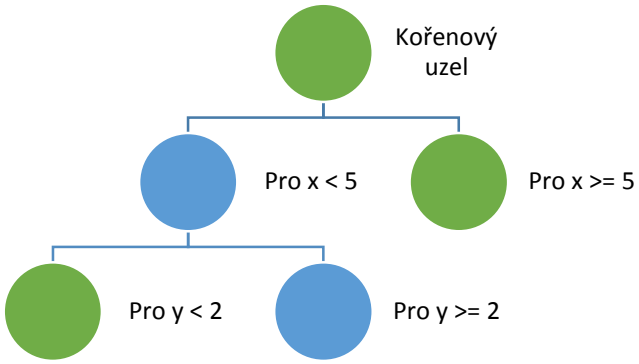
[1] Pro Framework Weka existuje implementace, která poskytuje na moderních strojích dle autorů 5-10x větší rychlost. Viz. http://code.google.com/p/fast-random-forest/ (dostupnost ověřena 10. 5. 2013)

5 Algoritmy založené na rozhodovacích stromech

Tato kapitola rozebírá datové struktury a algoritmy používané v rozhodovacích stromech. Nejprve jsou popsány znaky, které jsou společné pro algoritmy této skupiny, a následně zaměřuje na konkrétní algoritmy vybrané dle srovnání v předchozí kapitole.

Rozhodovací strom je graf stromové struktury, kde každý vnitřní uzel reprezentuje test hodnoty jistého atributu a koncové uzly (listy) reprezentují třídu, do které je daný objekt klasifikován. Takový rozhodovací strom může být snadno převeden na odpovídající klasifikační pravidla.

Následuje příklad možné podoby stromu. Mějme rovinu R, v níž každý bod je definován souřadnicí x a y. Dle souřadnice má bod přiřazenou barvu (třída). Příklad možného stromu můžeme vidět na obrázku 5.1.



Obrázek 5.1: Příklad stromu rozdělující rovinu (x y) dle barev

Většina klasifikačních algoritmů založených na rozhodovacích stromech vytvoří jeden výsledný strom. Jeho vznik můžeme rozdělit na dvě části – fáze růstu a ořezání.

Vstupem jsou data trénovací množiny S a seznam atributů L. Průběh fáze růstu může být popsán následujícím pseudokódem 5.1:

```
function CreateTree(S, L) : Tree
begin
    vytvoř uzel N
    if  (všechny s ∈ S jsou ve stejné třídě C) then
        return N jako list třídy C
    if  (isempty(L)) then
        return N jako list nejběžnější třídy v S
    vyber atribut A ∈ L s „nejvyšší rozhodovací schopností“ a odstraň jej
    z L
```

```
    nastav dělící atribut na A
    for (a_i ∈ A, kde i ∈ <1; sizeof(A)>) do
    begin
        vytvoř větev N_i z uzlu N pro všechna s_i ∈ S, kde hodnota A pro s_i
        je rovna a_i
        if (isempty(s_i)) then N_i je list s nejběžnější třídou v množině S
        else N_i := CreateTree(s_i, L)
    end
end
```

Výpis 5.1: pseudokód růstu stromu dle [3]

Tento kód je možné nalézt v literatuře v různých modifikacích v závislosti na použité datové struktuře. Velmi často se využívá například binárního stromu s tím, že dělení může pomocí jednoho atributu proběhnout vícekrát. V těchto situacích je třeba doplnit ještě omezení maximálního počtu dělení dle každého atributu.

Pro shrnutí: vytvoříme funkci určenou ke generování uzlu stromu, které předáme vzorky z trénovací množiny a také seznam atributů, na základě kterých můžeme provést rozvětvení stromu.

Zjistíme rozdělení vzorků do tříd – běžně ve formě histogramu – a pokud všechny vzorky patří do stejné třídy nebo je seznam atributů pro dělení prázdný, vrátíme uzel jako list označující třídu.

V ostaních případech projdeme všechny atributy v seznamu atributů k dělení a najdeme ten, který je nejvhodnější pro dělení. Způsobu výběru se budeme věnovat za chvíli.

Na základě zvoleného atributu rozvětvíme strom. Každá větev bude výsledkem rekurzivního volání funkce pro generování uzlu.

Jak bylo v předchozím textu upozorněno, je třeba vybrat vhodného kandidáta pro dělení. Pro to se používá nejčastěji kombinace dvou funkcí – *informačního zisku* a *míry entropie*, přičemž informační zisk je závislý na výpočtu míry entropie. Výběr je možný provést i pomocí jiných funkcí (např. gini index prezentovaný v kapitole 5.1). Pro další informace využijte [5].

Tyto dvě funkce je možné spočítat pomocí vzorců 5.1 a 5.2.

Informační zisk:

$$I(S,A) = H(S) - \sum_{i \epsilon a} \frac{|S^i|}{|S|} H(S^i)$$

Vzorec 5.1: Výpočet informačního zisku

kde:

S jsou vzorky z trénovací množiny,

A je atribut, pro který zjišťujeme zisk,

H(S) je hodnota Shannonovy entropie pro množinu *S*,

a představuje množinu hodnot $_{ai} \in A$,

$|S|$ je velikost množiny *S*.

Shannonova entropie:

$$H(S) = -\sum_{c \epsilon C} p(c) \log(p(c))$$

Vzorec 5.2: Shannonova entropie

kde:

S jsou vzorky z trénovací množiny,

C je množina všech tříd,

$c \in C$, kde c je konkrétní třída,

p(c) je pravděpodobnost výskytu třídy c.

Vybrán bude ten atritut, pro který je hodnota informačního zisku největší.

Během fáze růstu mohly vzniknout větve, které dělají stom složitějším a také méně kvalitním z hlediska předpovědi. Tyto větve je třeba ze stromu odstanit. Odstranění nežádoucích větví probíhá ve fázi *ořezání (prunning)* [1].

Ořezání stromu se dá řešit dvěma způsoby. První metodou je tzv. *prepruning* [1]. Při vytváření stromu se nežádoucí větve negenerují. Místo testu atributu na další podmínku se tato část stromu přímo ukončí listem, který se ohodnotí takovou klasifikační třídou, která je přiřazena nejvíce vzorkům odpovídají této části stromu.

Druhou metodou je tzv. *postpruning* [1]. Při něm je rozhodovací strom vytvořen celý. A teprve po jeho úplném vytvoření jsou „nepotřebné“ větve odstraněny. Další informace o ořezání je možné najít v [1], [2], [3].

5.1 SPRINT

Algoritmus SPRINT je následníkem algoritmu SLIQ. Využívají se v něm stejné principy, ale jsou odstraněny jeho nedostatky. V této části budou popsány nejdůležitější části algoritmu a rozdíly proti principům popsaným v přechozí části textu.

Principem algoritmu je vytvoření seznamu pro každý atribut, ve kterém jsou hodnoty seřazeny dle atributu. To může být paměťově velmi náročné, a proto může být seznam uložen na disk. Některé implementace využívají pouze seznamů uložených na disku, což negativně ovlivňuje rychlost. Proto se používá přístup, při němž jsou data dle své velikosti držena v paměti a v případě nedostatku paměti uložena na disk. [6]

V každém uzlu je uložen histogram s distribuční funkcí hodnot daného atributu.

Oproti základnímu algoritmu pro růst stromu není pro vyhodnocení dělícího atributu použit informační zisk, ale využívá se tzv. gini indexu. Ten je definován ve vzorci 5.3.

$$gini(S) = 1 - \sum_{c \in C} p(c)^2$$

Vzorec 5.3: výpočet gini indexu

kde:

S jsou vzorky z trénovací množiny,

C je množina všech tříd,

c ∈ C, kde c je konkrétní třída,

p(c) je frekvence výskytu třídy c.

Takto je ohodnocena původní datová množina a rozdělené množiny. Zároveň je třeba vypočíst gini index po rozdělení dle vzorce 5.4:

$$gini_{split} = \frac{|S^1|}{|S|} gini\left(S^1\right) + \frac{|S^2|}{|S|} gini\left(S^2\right)$$

Vzorec 5.4: výpočet gini indexu pro dělení

kde:

S jsou vzorky z trénovací množiny,

S^1, S^2 jsou množiny vznikle rozdělením,

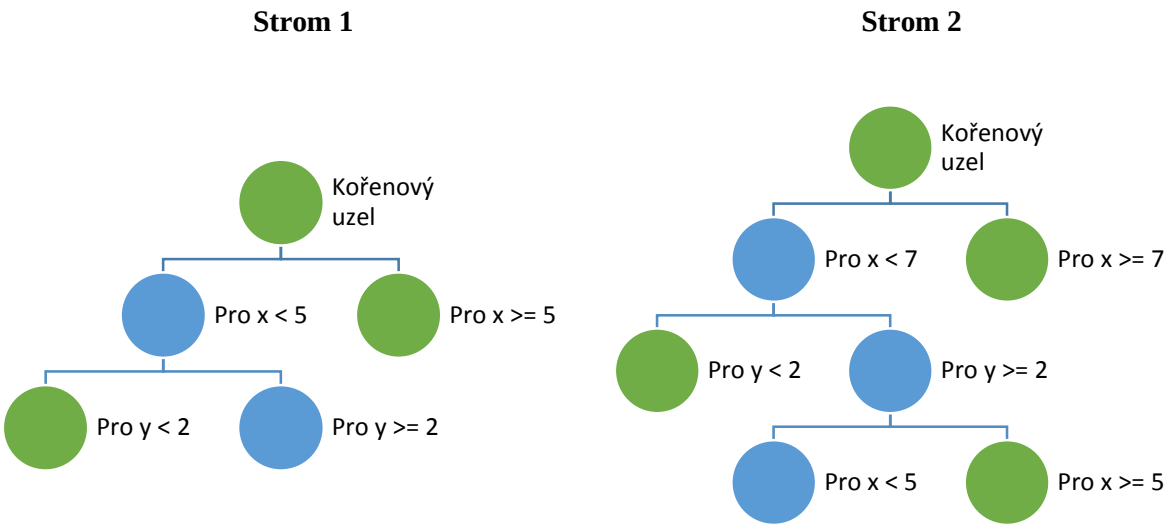
$|S|$ je velikost množiny *S.*

Pro rozdělení vybereme ten atribut, jehož gini index je nejmenší.

5.2 RANDOM FOREST

Tento algoritmus patří v současné době mezi nejpřesnější algoritmy založené na rozhodovacích stromech určené pro klasifikaci. Od ostatních algoritmů této kategorie se liší hned několika věcmi. Tou první je fakt, že se nejedná o strom, ale o les – tedy n stromů. Každý strom je trénován samostatně. Klasifikace je provedena všemi stromy a následně proběhne hlasování o výsledku. Díky tomu může dosahovat teoreticky lepších výsledků než 1 strom.

Ukážeme si, jak by mohl takový les vypadat pro příklad barev v prostoru ze začátku kapitoly.



Obrázek 5.2: Příklad RF pro 2 stromy – rozdělení roviny (x, y) dle barvy

Další důležitou vlastností je vložení náhodnosti do procesu tréninku. V této oblasti se používají dva odlišné přístupy.

Prvním z nich je tzv. *bagging* – pro každý strom je z trénovací množiny náhodně navzorkována podmnožina záznamů nad kterou následně probíhá trénink. Tato technika má výhodu ve zvýšení rychlosti tréninku vlivem menšího počtu vzorků. Také pomáhá překonat problémy, kdy se stromy mohou specializovat pouze na vybrané atributy. Nicméně nevýhodou je plýtvání trénovacími daty - některé záznamy nemusí být vůbec použity. [7]

Druhou technikou je *optimalizace v uzlu*. Při ní se z množiny atributů náhodně vybere podmnožina záznamů, která může být definovaná vzhledem k hloubce stromu nebo nastavena analytikem pomocí parametru algoritmu. Výhodou tohoto přístupu je možnost využít celou trénovací množinu. [8]

Výběr atributu k dělení je proveden stejně jako u výchozího algoritmu pomocí výpočtu míry informačního zisku a entropie [9].

Pro trénink stromu je možné využít dva přístupy – trénování do hloubky a trénování do šířky [10]. Při tréninku do hloubky je nejprve každý uzel trénován do dosažení maximální hloubky omezené parametrem, nebo dostatečné přesnosti klasifikace listu. Trénink od šířky nejdříve ve všech stromech trénuje uzly v aktuální hloubce.

Random Forest je velmi dobře paralelizovatelný a to jak ve fázi tréninku, tak i při klasifikaci. Vzhledem k paralelizaci je lepší využít přístupu trénování do hloubky, kdy každý strom necháme trénovat paralelně. Stejným způsobem vyřešíme i klasifikaci – tj. necháme paralelně klasifikovat záznam v každém stromě.

6 Analytický systém

Jedním z cílů práce je implementace algoritmů jako zásuvných modulů systému v rámci projektu „Systém pro zvýšení bezpečnosti v prostředí Internetu analýzou šíření škodlivého kódu“.V rámci projektu je vyvíjen systém pro dolování z dat určený pro *enterprise* nasazení. V této části bude stručně popsán systém, pro který jsou zásuvné modely tvořeny. Cílem projektu je vytvořit robustní, rozšiřitelný a vysoce výkonný systém, který podporuje získávání znalostí jak z uložených dat, tak i z datových proudů. Dalším z cílů je vytvoření sytému pro ukládání modelů získaných dolováním. Dále následuje popis jednotlivých cílů z širšího hlediska.

Základním požadavkem rozšiřitelnosti je umožnit rychlé nasazení nových, případně vylepšených, algoritmů pro dolování z dat. Z toho důvodu je definované API, které bude popsané v následující kapitole. V rámci projektu bude implementováno velké množství dolovacích algoritmů a nástroje pro vizualizaci výsledků.

Systém musí být schopný nepřetržitého běhu 24 hodin denně, 7 dní v týdnu. Cílem je tedy vytvoření systému vhodného do prostředí, kde je vyžadována vysoká úroveň dostupnosti a spolehlivosti. Systém musí zpracovat velké množství požadavků současně. Také se pracuje na podpoře online instalace a upgradu algoritmů. Tj. nový algoritmus musí být nasaditelný bez nutnosti restartu systému.

Cílem je podpora velkého množství zdrojů dat – ať už uložených v paměti, v perzistentním úložišti v relační databázi nebo *OLAP* kostce získaných *SQL* dotazem nebo *MDX/XMLA* dotazem.

Projekt je vyvíjen na modelu *System as a Service*. Serverová část běží jako služba na zařízeních s dostatečným výkonem. Klient k funkcím přistupuje pomocí webového rozhraní.

Systém je tvořený pod platformou .NET verze 4.0 v programovacím jazyce C#. To umožňuje multiplatformní nasazení (pod Windows se využívá Microsoft .NET). Zároveň také dává možnosti tvorby rozšíření v jakémkoliv programovacím jazyce podporovaném prostředím CLR.

Jednotlivé komponenty jsou vytvořené jako služby a komunikace mezi nimi je realizována pomocí frameworku Windows Communication Foundation. Detailnější popis možností platformy .NET a souvisejících technologií naleznete v [11].

Popisu API se věnuje kapitola 7, zaměřená na implementaci zásuvných modulů. Další informace o projektu lze nalézt v [12].

7 Implementace vybraných algoritmů

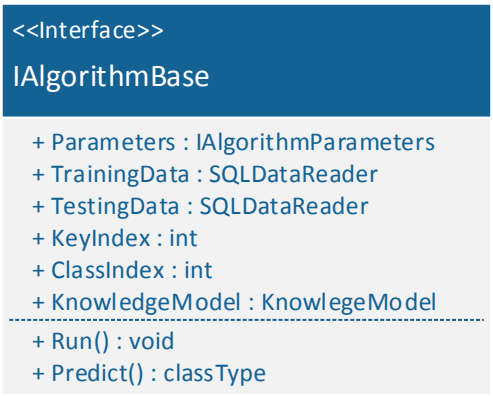
Tato část se zabývá implementací jednotlivých algoritmů a zásuvných modulů pro analytický systém popsaný v kapitole 6. Implementační jazyk byl zvolen dle systému, stejně tak jako jeho verze. Konkrétně se jedná o jazyk C# 2010 pod platformou .NET verze 4.0. Nejprve je popsáno, jak vypadá API analytického systému, pro který byly vytvořeny moduly a jaké musely být dodrženy požadavky na jejich návrh. Následně jsou popsány datové struktury, které byly využity pro implementované algoritmy a na závěr kapitoly jsou popsány detaily implementace jednotlivých algoritmů.

Kapitola obsahuje schémata diagramu tříd a výpisy pseudokódu popisující implementační detaily zvolených řešení. Diagramy tříd jsou tvořeny na základě pravidel popsaných ve zdroji [13]. Pseudokód odpovídá syntaxi jazyka C# pro co možná nejlepší orientaci při porovnání se zdrojovým kódem.

7.1 API analytického systému

V rámci systému existují dva důležité jmenné prostory. Atributy a metody jsou ponechány bez dalšího popisu vzhledem k jednoznačnému odvození jejich účelu z názvu. Prvním je prostor `MAS.AlgorithmInterface`, kde jsou definována rozhraní, která musí implementovat dolovací algoritmus.

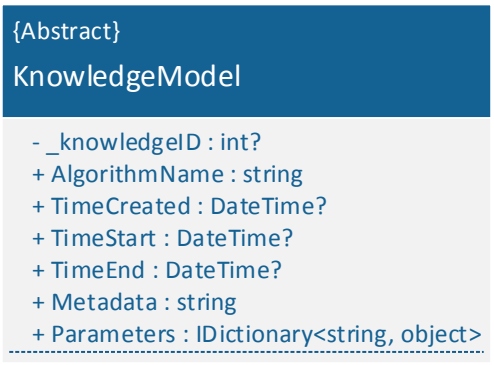
Prvním z rozhraní je `IAlgorithmBase` ukázaný na následujícím diagramu tříd.



Obrázek 7.1: Diagram rozhraní `IAlgorithmBase`

Další rozhraní definované ve jmenném prostoru `MAS.AlgorithmInterface` je `IAlgorithmParameters`, které slouží pro uložení konfiguračních parametrů dolovacího algoritmu. V tomto rozhraní nejsou definované žádné atributy nebo metody. Implementace potřebných atributů je závislá pouze na potřebách autora zásuvného modulu.

Druhý jmenný prostor definující podobu zásuvného modulu je `MAS.KnowledgeModels`. V tomto prostoru najdeme abstraktní třídu `KnowledgeModel`, která popisuje model dat dolovacího algoritmu určeného pro trénink, klasifikaci nebo i vizualizaci. Její podoba je zachycena na diagramu 7.2.



Obrázek 7.2: Diagram abstraktní třídy `KnowledgeModel`

V této části jsme ukázali rozhraní, které musíme dodržet pří vytvoření zásuvných modulů pro analytický systém. Dále se již budeme soustředit na podstatné detaily implementace dolovacích algoritmů a ponecháme nepodstatné části implementace na prozkoumání zdrojového kódu čtenářem.

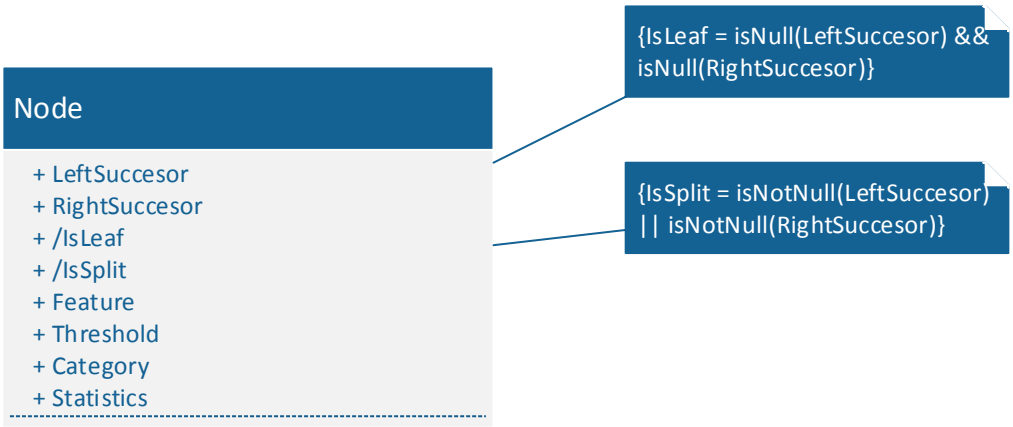
7.2 Datové struktury používané pro algoritmy

V rámci první fáze implementace bylo třeba vytvořit datové struktury, které budou využity pro vybrané dolovací algoritmy. Základní strukturou, která bude třeba, je strom. V této prací byla zvolena varianta binárního stromu z důvodu snadné implementace jak struktury, tak i její využití v algoritmech. Detailní informace k datovým strukturám a jejich implementaci je možné nalézt v [14].

Binární strom je tvořen dvěma třídami - třídou `Tree` a třídou `Node`.

7.2.1 Datová struktura uzlu

Třída `Node` obsahuje atributy popisující jeden uzel.



Obrázek 7.3: Diagram třídy `Node`

Každý uzel může mít nejvýše dva následníky (jedná se o uzel binárního stromu). Tyto následníky reprezentuje property `LeftSuccesor` a `RightSuccesor`.

Na základě následnických uzlů můžeme říci, zda se jedná o list (property `IsLeaf`) nebo o dělící uzel (`IsSplit`).

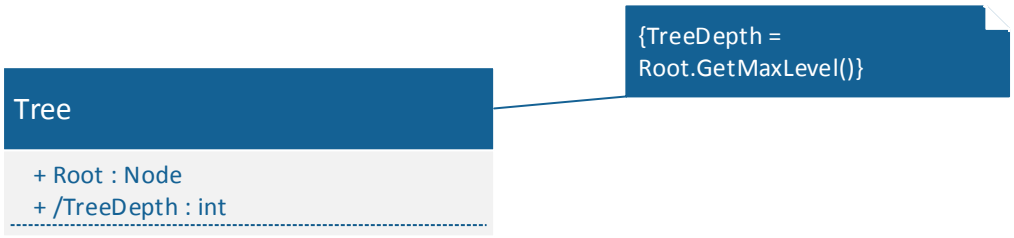
Pokud se jedná o dělící uzel, je v property `Feature` nastaven atribut podle kterého probíhá dělení a také property `Threshold`, ve které je uložena prahová hodnota pro dělení.

V případě, že se jedná o list, je nastavena property `Category` na základě histogramu uloženého v property `Statisctics`. Výsledek pro klasifikaci je ukládán v každém uzlu a ne pouze v listech jako u většiny implementací. To umožňuje provést testy s jinými parametry algoritmu bez nutnosti nového učení (příkladem může být zmenšení maximální hloubky stromu). Zároveň u algoritmu Random Forest, kde záleží na pravděpodobnosti, je srovnání výsledků při jiném nastavení parametrů relevantnější.

7.2.2 Datová struktura stromu

Binární strom je reprezentován třídou `Tree`. Tato třída obsahuje kořenový uzel (property `Root`) a property vracející hloubku stromu. Samostatná třída pro reprezentaci stromu byla zvolena z důvodu pozdějšího přidání metod, které by pracovali s celým stromem a byl dodržen princip *SoC* (*Separation of Concerns*). SoC říka, že program musí být rozdělen tak, aby každá z jeho částí měla pouze jednu odpovědnost – viz. [15], [16].

Podoba třídy `Tree` je zachycena v diagramu na obrázku 7.4.

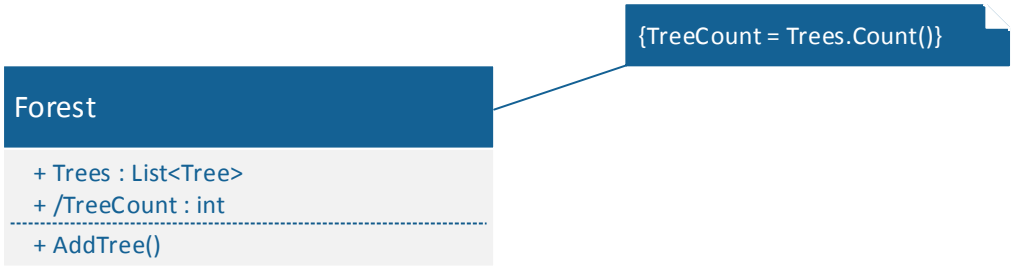


Obrázek 7.4: Diagram třídy Tree

7.2.3 Datová struktura lesa

Pro potřeby algoritmu Random Forest byla implementována třída `Forest`. Tato třída zapouzdřuje všechny vytvořené stromy. Jejich seznam je dostupný přes property `Trees`. Mezi další důležité komponenty této třídy patří property vracející velikost lesa (počet jeho stromů), a metoda pro přidání dalšího stromu do lesa.

Výsledná třída `Forest` je zobrazena na obrázku 7.5.

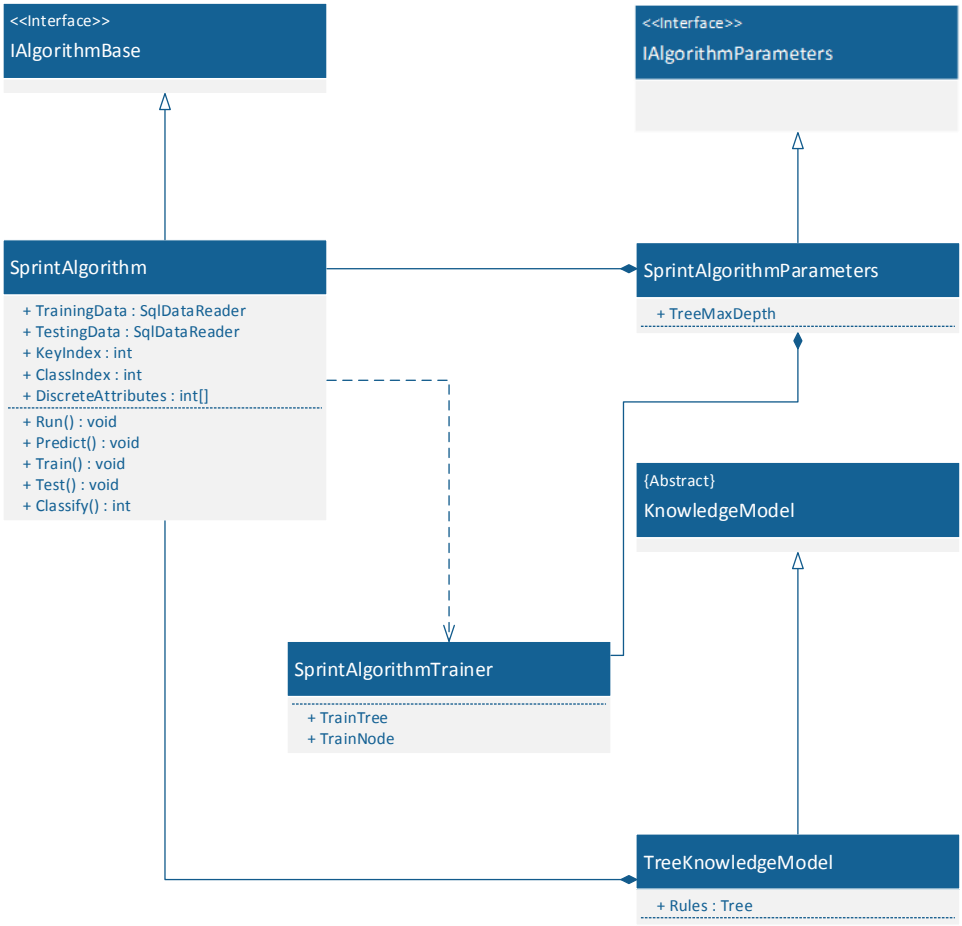


Obrázek 7.5: Diagram třídy `Forest`

Po dokončení implementace datových struktur bylo možné přejít na implementaci samotných algoritmů. V následující části je rozebráno, jakým způsobem byly v rámci této bakalářské práce vytvořeny zásuvné moduly s algoritmy.

7.3 SPRINT

Algoritmus SPRINT byl implementován dle diagramu na obrázku 7.6. Je využita hybridní sériová implementace popsaná v [6].



Obrázek 7.6: Diagram tříd algoritmu SPRINT

Nejprve byla vytvořena třída `SprintAlgorithm` (implementující rozhraní `IAlgorithmBase`), `SprintAlgorithmParams` (implementující rozhraní `IAlgorithmParams`) a třída `TreeKnowledgeModel` (implementující abstraktní třídu `KnowledgeModel`).

Třída `SprintAlgorithm` zodpovídá za veškerou činnost prováděnou algoritmem. Konkrétně její metody:

- `Run()` – spouští proces tréninku a následného testování s využitím metod `Train()` a `Test()`.
- `Predict()` – provádí klasifikaci na zadaném vstupním proudu dat.
- `Train()` – stará se o vytvoření znalostního modelu s využitím trenéra (třída `SprintAlgorithmTrainer`) a vstupní množiny `TrainingData`.
- `Test()` – provede testování znalostního modelu s využitím metody `Classify()` a vstupní množiny `TestingData`.

Před zahájením činnosti třídy musíme nastavit parametry trénovacího algoritmu. V tomto případě se jedná pouze o atribut omezující maximální hloubku stromu.

Kromě parametrů je třeba stavit zdroj trénovacích dat a zdroj testovacích dat. Zdroj dat musí být instancí třídy `SqlDataReader`. Trénovací data ukládáme do property `TrainingData` a testovací do `TestingData`.

Následně je třeba u každého algoritmu nastavit, který atribut v datové množině představuje identifikátor (`KeyIndex`), třídu (`ClassIndex`), a které atributy nabývají diskrétních hodnot (`DiscreteAttributes`). Diskrétní atributy je třeba pro dosažení lepších výsledků zpracovávat jiným způsobem než atributy spojité.

Když je provedeno veškeré nutné nastavení, je možné přejít na popis hlavní činnosti algoritmu. Začneme metodou `Train()`. V metodě `Train()` je vytvořena instance třídy `SprintAlgorithmTrainer`, které slouží k vytvoření znalostního modelu. Před zahájením tréninku pokračujeme načtením dat trénovací množiny.

Nad instancí trenéra je vyvolána metoda `TrainTree()`, které je předána trénovací množina. Výsledek je použit pro vytvoření znalostního modelu.

Nyní se podíváme na proces tréninku stromu. V rámci metody `TrainTree()` je pouze vytvořeno pole seznamů seřazených dle atributů. Toto pole je předáno metodě `TrainNode()`, která má na starost trénování uzlů stromu. `TrainNode()` nejprve vytvoří instanci uzlu a vytvoří histogram (třída `Histogram`), který zodpovídá za výběr výsledné třídy pro uzel. Uzlu je nastavena nejčastěji se vyskytující třída. Pokud všechny prvky z aktuální trénovací množiny patří do jedné třídy, je uzel vrácen jako list. Jinak jsou postupně testovány všechny atributy pro získání nejhodnějšího kandidáta pro dělení.

Výběr kandidáta je popsán na algoritmu 7.2 na konci kapitoly. Na základě nejlepšího kandidáta a vypočítané prahové hodnoty je provedeno dělení trénovací množiny. Dělení u algoritmu SPRINT spočívá pouze ve výběru správně části ze seznamu hodnot atributů. Pro levou a pravou část trénovací množiny je rekurzivně volána metoda `TrainNode()`, jejíž výsledky jsou připojeny jako levý (resp. pravý) následník aktuálního uzlu.

Zjednodušená implementace je popsána na algoritmu 7.1.

```
class SprintAlgoritmTrainer {
     public virtual void TrainNode(AttributeList trainingSet) {
          Node n = new Node();
          Histogram h = new Histogram(trainingSet);
          int mostFrequent = h.GetMostFrequent();
          Node.Category = mostFrequent;
          if (h.GetProbability(mostFrequent) == 1.0) return n;
```

```
            int splitAttribute = GetBestSplitAtrribute(attributeList);
            AttributeList leftTrainingSet =
                    GetLeftDataSet(splitAttribute);
            AttributeList rightTrainingSet =
                    GetRightDataSet(splitAttribute);
            n.LeftSuccesor = TrainNode(leftTrainingSet);
            n.RightSuccesor = TrainNode(rightTrainingSet);

            return n;
        }
    }
```

Algoritmus 7.1: pseudokód metody `TrainNode()`

Následující pseudokód popisuje výběr atributu pro dělení:

```
public virtual void Train(AttributeList attributeList) {
    float threshold = CountThreshold(attributeList);
    double gini = double.MaxValue;
    int attribute = -1;
    foreach (var candidate in attributeList) {
        AttributeList leftTrainingSet =
                GetLeftDataSet(candidate);
        AttributeList rightTrainingSet =
                GetRightDataSet(candidate);
        Histogram h1 = new Histogram(leftTrainingSet);
        Histogram h2 = new Histogram(rightTrainingSet);

        if (gini > CountGiniIndex(h1, h2)) {
          attribute = candidate;
          gini = CountGiniIndex(h1, h2)
        }
    }
    return attribute;
}
```

Algoritmus 7.2: pseudokód výběru nejvhodnějšího kandidáta pro dělení

Metoda `Test()` pouze využívá metodu `Classify()` a shromažďuje statistiku úspěšnosti. Není třeba ji zde podrobněji rozbírat. Případný zájemce má možnost nahlédnout do zdrojového kódu algoritmu.

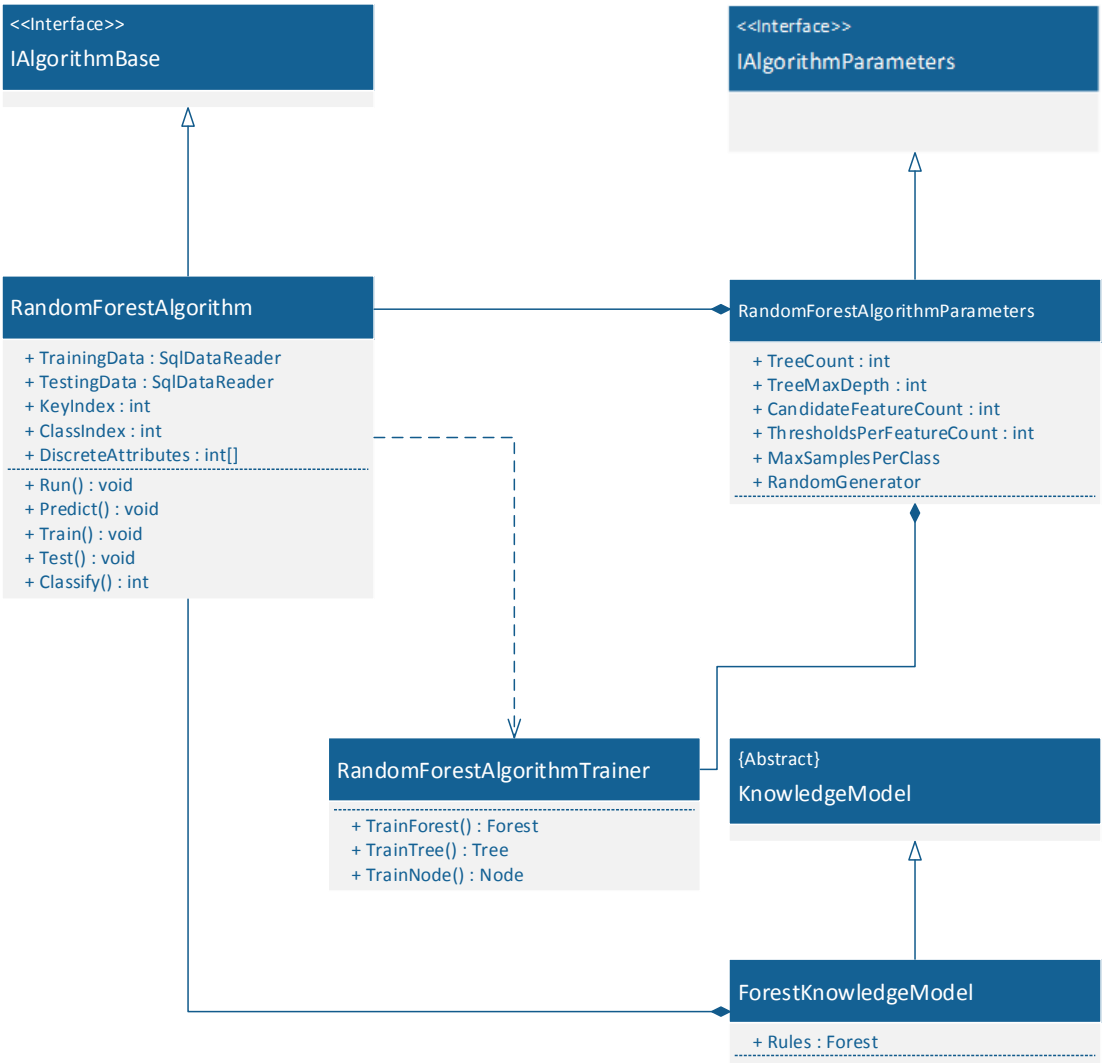
Další důležitou částí je metoda `Classify()`. Ta prochází binární strom způsobem *preorder*. Jakmile narazí na list, vrátí výsledek klasifikace. Vzhledem k jednoduchosti implementace zde pseudokód není uveden. Čtenář má možnost nahlédnout do zdrojového kódu programu.

V této části jsme prošli implementaci algoritmu SPRINT. Jeho nejdůležitější částí je využití seznamu atributů (existence seřazeného seznamu hodnot pro každý atribut) a způsob volby kandidáta pro dělení. Přístup algoritmu SPRINT je paměťově velmi náročný. V základní implementaci je proto seznam atributů uložen na disku. V případě hybridní implementace je držen v paměti, pokud je to možné. U velkých databází jsme zde limitování implementací seznamu na 2GB dat. Toto omezení je možné obejít implementací jiné datové struktury nebo využitím .NET verze 4.5 a konfigurací *garbage collectoru* pro velké objekty (umožní využít pole větší než 2GB – implementace základního seznamu v .NETu je řešená pomocí pole).

7.4 RANDOM FOREST

Jak bylo zmíněno dříve, zásadním rozdílem oproti jiným algoritmům je ten, že Random Forest využívá náhodnosti při tvorbě stromu, a místo jednoho stromu jich tvoří několik. Těmto hlavním aspektům se věnuje tato kapitola.

Architektura modulu implementujícího Random Forest je možné vidět na obrázku 7.7.



Obrázek 7.7: Diagram tříd algoritmu `Random Forest`

O řízení algoritmu se stará třída `RandomForestAlgorithm`. Instanci této třídy je třeba nastavit stejně jako odpovídající třídu implementace algoritmu Sprint. Pro detaily týkající se popisu jednotlivých atributů je možné nalézt v kapitole 7.3. Jediné dvě části, které se liší, jsou metody `Train()` a `Classify()`. Nejprve je zde popsána metoda `Train()` a její způsob vytvoření znalostního modelu, a následně metoda `Classify()` a způsob predikce dat.

Metoda Train() nejprve vytvoří instanci třídy RandomForestAlgorithmTrainer a načte data z trénovací množiny. Trénovací množinu předá při volání metody TrainForest(). V rámci metody TrainForest() je paralelně volána metoda TrainTree(). K paralelizaci byla využita paralelní implementace cyklu for, kterou poskytuje .NET [11]. Abychom mohli použít paralelní variantu, není možné použít pro generování náhodných čísel třídu System.Random, jelikož by pro všechna vlákna vracela stejná náhodná čísla. Proto v práci byl implementován *thread safe* generátor náhodných čísel (třída ParallelRandomGenerator).

Před zahájením trénování stromu se zkontroluje parametr MaxSamplesPerClass, jestli je třeba provést navzorkování vstupních dat pro každý strom (viz. metoda *bagging* popsaná v kapitole 7.2). Stromu je poté předána jeho vlastní trénovací množina. Metoda TrainTree() inicializuje kořenový uzel a zajistí trénování jak tohoto uzlu, tak jeho potomků.

Metoda TrainNode() si nejprve vytvoří histogram pro aktuální trénovací množinu. Na základě tohoto histogramu přiřadí statistiky uzlu a nastaví výslednou třídu. V případě, že všechny prvky množiny patří do stejné třídy, vrátí uzel jako list. Algoritmus metody TrainNode() odpovídá implementaci na výpisu 7.1. Rozdílný je ovšem výběr atributů pro dělení. Místo gini indexu je v Random Forestu použit informační zisk a entropie. Je použit ten atribut, jehož informační zisk je největší. Proces výběru atributu pro dělení je popsán na algoritmu 7.3.

```
public virtual void Train(AttributeList attributeList) {
     float threshold = CountThreshold(attributeList);
     double gain = double.MinValue;
     int attribute = -1;
     foreach (var candidate in attributeList) {
          AttributeList leftTrainingSet =
                  GetLeftDataSet(candidate);
          AttributeList rightTrainingSet =
                  GetRightDataSet(candidate);
          Histogram h1 = new Histogram(leftTrainingSet);
          Histogram h2 = new Histogram(rightTrainingSet);

          if (gain <= CountInformationGain(h1, h2)) {
            attribute = candidate;
            gain = CountInformationGain (h1, h2)
          }
     }
     return attribute;
}
```

Algoritmus 7.3: Pseudokód výběru atributu pro dělení u algoritmu Random Forest

8 Experimenty s implementovanými algoritmy

V rámci této kapitoly jsou prováděny s implementovanými algoritmy experimenty s různými parametry. Pro každý experiment jsou nejprve popsány nastavené parametry. Do výsledkové tabulky byly zaneseny hodnoty udávající přesnost algoritmu a dobu učení při konkrétním nastavení. Výsledky jsou porovnány a na jejich základě jsou vyvozena doporučení pro uživatele.

8.1 Body v rovině

Pro experiment body v rovině byla vygenerována databáze obsahující 2 třídy a 1 000 vzorků trénovací množiny. Prostor byl rozdělen v první třetině a do každé části vygenerováno 500 vzorků. Vzorky vlevo byly označeny barvou 1 a vzorky vpravo barvou 2. Pro oba algoritmy byla úspěšnost klasifikace 100% a doba učení byla v řádu desítek milisekund. Experiment tedy neposkytl výsledky vhodné pro srovnání.

Druhým experimentem tedy bylo upravení databáze. Byla vygenerována databáze čítající 20 000 vzorků a 4 třídy. Prostor byl rozdělen na 4 obdélníky – 2 nahoře a 2 dole. Algoritmus SPRINT měl ihned výsledek 100%, algoritmus Random Forest po nastavení parametrů na 7 stromů a hloubku 8 taktéž. Časy tréninku byly v řádu stovek milisekund. Ani u druhého experimentu nebyly výsledky vhodné pro srovnání.

Pro zvýšení doby potřebné pro trénink tedy bylo nutné přistoupit k vygenerování ještě větší databáze a rozdělení prostoru na větší množství útvarů. Řešení je popsané v kapitole 8.2.

8.2 Body v prostoru

V rámci tohoto experimentu byla vygenerována databáze čítající 120 000 vzorků trénovací množiny. Přibyla další dimenze (z) a prostor byl rozdělen na 24 částí, kterým byla náhodně přidělena barva a vygenerováno uvnitř nich náhodně 5 000 vzorků.

V tabulce 8.1 jsou popsány výsledky algoritmu SPRINT a v tabulce 8.2, 8.3 výsledky algoritmu Random Forest pro Criminisiho metodu resp. Breimanovu. V tabulkách je zároveň uvedeno použité nastavení pro testy.

Číslo experimentu	Hloubka stromu	Chybovost (%)	Rychlost (s)
1	5	79,41%	1,5501s
2	10	76,81%	2,4933s
3	20	1,20%	4,9756s
4	30	0,27%	5,0666s
5	60	0,27%	5,0656s
6	22	0,27%	4,9652s
7	21	0,32%	4,9726s

Tabulka 8.1: Výsledky algoritmu SPRINT

Číslo experimentu	Počet stromů v lese	Hloubka stromu	Kandidátních atributů	Chybovost (%)	Rychlost (s)
1	3	10	1	75,13%	0,5191s
2	50	10	1	73,37%	5,2746s
3	10	20	1	0,66%	9,5422s
4	20	20	1	0,59%	21,3857s
5	30	20	1	0,44%	32,4021s
6	10	25	1	0,26%	10,6353s
7	10	30	1	0,26%	10,9643s
8	10	20	2	0,61%	13,7227s

Tabulka 8.2: Výsledky algoritmu Random Forest (klasifikace histogramem)

Číslo experimentu	Počet stromů v lese	Hloubka stromu	Kandidátních atributů	Chybovost (%)	Rychlost (s)
1	10	20	1	1,78%	9,2401s
2	10	30	1	0,26%	11,1414s
3	30	20	1	1,04%	33,0642s

Tabulka 8.3: Výsledky algoritmu Random Forest (klasifikace třídou v uzlu)

8.3 Přeživší na Titanicu

Server Kaggle[2] nabízí řadu zajímavých datových sad v rámci data miningových soutěží. Pro další experiment byla využita datová sada, jejímž cílem je určit, kdo přežije na Titanicu. Trénovací množina

[2] Kaggle se zabývá soutěžemi v data miningu. Je dostupný na adrese www.kaggle.com

má 900 vzorků, testovací má 440 vzorků. Datové sady byly diskretizovány a vloženy do databáze. Původní i diskretizovaná sada jsou dostupné na přiloženém CD.

Experimenty a použité parametry algoritmů je uvedeny v tabulce 8.4 a 8.5.

Číslo experimentu	Hloubka stromu	Chybovost (%)	Rychlost (s)
1	5	14,11%	0,0620s
2	10	18,18%	0,0650s
3	20	18,18%	0,0880s
4	30	20,09%	0,1060s
5	4	14,59%	0,0560s

Tabulka 8.4: Výsledky algoritmu SPRINT

Číslo experimentu	Počet stromů v lese	Hloubka stromu	Počet atributů pro dělení	Chybovost (%)	Rychlost (s)
1	20	4	1	21,05%	0,2105s
2	20	5	1	21,29%	0,0400s
3	20	5	2	18,78%	0,0440s
4	20	5	3	19,37%	0,0590s
5	20	10	2	11,00%	0,1100s
6	20	20	2	8.37%	0,1410s
7	30	30	2	9,33%	0,1720s
8	100	20	2	7,41%	0,6150s
9	1000	20	2	7,17%	5,5236s

Tabulka 8.5: Výsledky algoritmu Random Forest

8.4 Hodnocení přesnosti algoritmů

V experimentu 8.2 byly schopné oba algoritmy dosáhnout podobných výsledků z hlediska přesnosti. U algoritmu Random Forest velmi záviselo na nastavených parametrech. Vzhledem k trénovací množině se jako ideální ukázala hloubka 25, což je dáno povahou dat (počtem oblastí v prostoru). Za zajímavé je možné považovat srovnání, kdy vrácení výsledku uzlu formou histogramu (způsob navržený A. Criminisim [10]) mělo při stejném nastavení přesnější výsledky klasifikace než vrácení pouze výsledné třídy. Výsledky lze dorovnat zvýšením počtu stromů, ale za cenu nárůstu času. Z těchto důvodů bylo dále použito klasifikování pomocí histogramu.

8.5 Zhodnocení rychlosti algoritmů

Implementace algoritmu SPRINT se ukázala na experimentech rychlejší než algoritmus Random Forest. Toto je dáno principem algoritmů. SPRINT tvoří pouze jeden strom, zatímco algoritmus Random Forest několik. Výsledky by nebyly tolik rozdílné v případě, že by algoritmus Random Forest měl k dispozici více jader procesoru, jelikož implementace v této práci je téměř lineárně škálovatelná s počtem jader. V případě využití *baggingu*, je možné algoritmus Random Forest urychlit.

Doba potřebná pro trénování algoritmu Random Forest roste lineárně spolu s počtem stromů a zároveň lineárně s hloubkou stromu. Pro algoritmus SPRINT doba učení roste také lineárně s počtem stromů.

8.6 Vliv nastavení parametrů na výsledky

Tato část popisuje vliv parametrů na výsledky algoritmů. V kapitole 8.6.1 jsou diskutovány výsledky jak algoritmu SPRINT, tak Random Forestu. Kapitoly 8.6.2 a 8.6.3 se zabývají pouze algoritmem Random Forest, jelikož probírané parametry se nevztahují k algoritmu SPRINT.

8.6.1 Hloubka stromu

Různá hloubka stromu ovlivňuje přesnost klasifikace. Do určité úrovně přesnost roste a následně začíná klesat. U algoritmu SPRINT dochází pouze k růstu přesnosti a následně se hodnota drží na konstantní úrovni. To je dáno fází ořezání stromu a dalšími vnitřními mechanismy. U algoritmu Random Forest fáze ořezání chybí, tudíž lze pozorovat snížení přesnosti klasifikace při dosažení určité úrovně. U obou algoritmů roste s hloubkou stromu lineárně čas potřebný k učení.

8.6.2 Velikost lesa

Každý strom lesa se podílí na klasifikaci výsledku. V případě, že jejich hlasem je histogram, je využita i váha hlasu. Počet stromů ovlivňuje výsledky více u klasické varianty, kdy list vrací jako výsledek konkrétní třídu. Přesnost roste s počtem stromů v lese. To je vykoupeno lineárním růstem času potřebným pro vytvoření znalostního modelu. Je třeba tedy vždy zvolit optimální poměr rychlosti a přesnosti výsledků.

8.6.3 Počet kandidátů pro dělení

Posledním parametrem, na kterém závisí výsledky, je počet kandidátních atributů pro dělení. Výsledky ukázaly, že počet kandidátů musí být malý oproti celkovému počtu atributů. S rostoucím počtem kandidátů se snižuje náhodnost a stromy jsou si více podobné ne-li totožné.

9 Závěr

Jádrem této práce bylo zvolit data miningové algoritmy založené na rozhodovacích stromech určených pro klasifikaci, implementovat vybrané algoritmy jako zásuvné moduly pro analytický systém vyvíjený v rámci projektu „Systém pro zvýšení bezpečnosti v prostředí Internetu analýzou šíření škodlivého kódu“ a ukázat nejdůležitější části implementace těchto algoritmů.

Na základě kritérií popsaných v kapitole 4 byly vybrány algoritmy SPRINT a Random Forest, a to z důvodů přesnosti jejich výsledků, možnosti paralelizace a zajímavosti implementace.

V kapitole 5 se čtenář mohl seznámit s myšlenkami, na kterých jsou algoritmy postaveny a následně v kapitole 7 projít implementační detaily zajímavých části algoritmů.

Výsledkem implementace byly zásuvné moduly pro zmíněný analytický systém. Implementace algoritmu Random Forest byla provedena s ohledem na maximální využití paralelizace a tím pádem zkrácení času potřebného pro učení a klasifikaci.

Algoritmy je možné konfigurovat. To nám může pomoci dosáhnout požadovaných výsledků. Zvláště algoritmus Random Forest je ovlivnitelný velkým množstvím parametrů. Nejdůležitějšími z nich je maximální hloubka stromu a počet stromů v lese. Na experimentech jsme ukázali, že příliš velká hloubka stromu vede k přeučení. S počtem stromů v lese zase narůstá přesnost klasifikace. Cenou za zvýšení přesnosti je ale lineární růst času potřebného pro vytvoření znalostního modelu a klasifikaci, což se dá omezit využitím více jader.

Dalším vylepšením této práce by mohlo být rozšíření, které umožní analytikovi využít i jiných útvarů pro dělení, než jsou hyperkrychle, a dát algoritmu možnost volby algoritmu pro výpočet prahové hodnoty pro dělení. Tyto úpravy by pomohli v dosažení přesnějších výsledků klasifikace.

Literatura

[1] **ZENDULKA, doc. Ing. Jaroslav , a další.** *Získávání znalostí z databází.* Brno : Fakulta informačních technologií VUT, 2009.

[2] **HAN, Jiawei, KAMBER, Micheline a PEI, Jian.** *Data Mining: Concepts and Techniques, Third Edition.* 3. vydání. místo neznámé : Morgan Kaufman, 2011. ISBN 978-0123814791.

[3] **WITTEN, Ian H., FRANK, Eibe a HALL, Mark A.** *Data Mining: Practical Machine Learning Tools and Techniques.* Third Edition. Burlington : Morgan Kaufmann, 2011. str. 664. ISBN 978-0123748560.

[4] **VENKATADRI, M a LAKANATHA, Reddy C.** A COMPARATIVE STUDY ON DECISION TREECLASSIFICATION ALGORITHMS IN DATA MINING. *International Journal of Computer Applications in Technology.* Apr - Sep, 2010, Sv. 2, 2.

[5] **DUNHAM, Margaret H.** *Data Mining: Introductory and Advanced Topics.* Upper Saddle River : Prestice Hall, 2002. ISBN 9780130888921.

[6] **SHAFER, R., AGRAWAL, R. a MEHTA, M.** *SPRINT: A Scalable Parallel Classifier for Data Mining.* London : Springer, 1996. ISSN 1066-8888.

[7] **BREIMAN, Leo.** *Bagging predictors.* Berkeley : Statistics Department, University, 1994.

[8] **ROBINSON, Andrew P. a HAMANN, Jeff D.** *Forest Analytics with R: An Introduction.* London : Springer, 2010. ISBN 978-1441977618.

[9] **BERG, Richard A.** *Statistical Learning from a Regression Perspective.* London : Springer, 2008. ISBN 978-0387775005.

[10] **CRIMINISI, Antonio.** *Decision Forests for Computer Vision and Medical Image Analysis.* London : Springer, 2013. str. 387. ISBN 978-1447149286.

[11] **TROELSEN, Andrew.** *Pro C# 2010 and the .NET 4 Platform.* New York : Apress, 2010. ISBN 978-1430225492.

[12] **KUPČÍK, Jan a HRUŠKA, Tomáš.** *Towards Online Data Mining System for Enterprises.* Wrocław : SciTePress - Science and Technology Publications, 2012. ISBN 978-989-8565-13-6.

[13] **FOWLER, Martin.** *Destilované UML.* Praha : Grada, 2009. ISBN 978-80-247-2062-3.

[14] **KARUMANCHI, Narasimha.** *Data Structures and Algorithms Made Easy: Data Structure and Algorithmic Puzzles, Second Edition.* Ney York : CreateSpace Independent Publishing Platform, 2011. ISBN 978-1468108866.

[15] **MCCONNELL, Steve.** *Code Complete: A Practical Handbook of Software Construction.* Redmont : Microsoft Press, 2004. ISBN 978-0735619678.

[16] **MICROSOFT PATTERNS & PRACTICES TEAM.** *Microsoft® Application Architecture Guide.* Redmont : Microsoft Press, 2009. ISBN 978-0735627109.

Seznam příloh

Příloha A – obsah CD

Příloha A

Obsah CD

- /src/ - zdrojové kódy aplikace
- /bin/ - přeložená aplikace
- /examples/ – datová sada
- /doc/ - tato práce ve formátu .docx a .pdf